Reading and writing CDMS-incompatible ASCII or binary files

Outline

- ASCII input files
- Binary input files
- Writing data to output files

Some grounding

- Python itself, especially via the ***string*** module, makes it really easy to manipulate string, and therefore ingest ASCII data.

- The ***struct*** module, coupled with the ***Numeric*** package allows for some ingestion of strictly binary files.

Reading text files in Python

- In its simplest form python provides useful tools to read ASCII data via string manipulation:

```
>>> f=open('file.txt')
>>> lines=f.readlines()
>>> for ln in lines:
...     sp=ln.split()
...     print ln, "splits to:", sp
```

------------ Example output from above ---------------

"o3, 0.3462, 0.5834" splits to: ["o3", "0.3462", "0.5834"]

"no2, 2.4435, 3.4352" splits to: ["no2", "2.4435", "3.4352 "]

ASCII files via contrib package: asciidata

- The contributed **asciidata** module provides some simple ASCII file reading functions.
- Imagine a file containing tab_delimited data:

```
var1    var2
22      44.3
34      48.3
```

```
>>> import asciidata
>>> a=asciidata.tab_delimited('tab_del_data.txt')
>>> print a
{'var1': array([ 22.,  34.]), 'var2': array([ 44.3,
  48.3])}
```

ASCII using VCDAT's browser module (1)

- The ASCII file reading capabilities of VCDAT can be accessed from the command line via the "browser" module.
- For non-formatted data:

```
browser.gui_ascii.read(text_file ,header=0,
   ids=None, shape=None, next='------
   ',separators=[';',',',':'])
```

- **header**: number of lines to skip at the beginning of the file
- **ids**: Name(s) to assign to the variables returned
- **shape**: Shape(s) to give to each variable read
- **next**: string separator between each variable
- **separators**: string separating elements

ASCII using VCDAT's browser module (2)

- For data in columns:

  ```
  browser.gui_ascii_cols.read( text_file ,header=0,
     cskip=0, cskip_type='columns', axis=0, ids=None,
     idrow=0, separators=[';',',', ':'])
  ```

- **cskip**: number of column/character to skip
- **cskip_type**: what to skip column or character
- **axis**: 0/1 is the first column to be used as an axis for the 1D variables
- **idrow**: 0/1 use the first row to set variable ids
- **ids**: name to give to variables returned

ASCII files via contributed package: Scientific (1)

You can read ASCII “Fortran Formatted” files using the Scientific contributed package:

```
>>> f = open(ascii_filename, 'r')
>>> # Import the module that does the work.
>>> from Scientific.IO import FortranFormat
>>> # Declare the fortran formats used to create the
>>> # data.
>>> ff1 = FortranFormat.FortranFormat('2i6')
>>> ff2 = FortranFormat.FortranFormat('12i6')
```

ASCII files via contributed package: Scientific (2)

```
>>> data_line = f.readline()
>>> mon,yr=FortranFormat.FortranLine(data_line,ff1)
>>> # Now define an array to read the data into.
>>> import Numeric
>>> T_array = Numeric.zeros((14,))
>>> # In the next line you are assigning the values.
>>> T_array[start_index: end_index] = \
   FortranFormat.FortranLine(f.readline(), ff2)
>>> # Note:You must have previously defined T_array.
>>> # See tutorial examples for more details.
```

Reading Binary files

- Fortran code also produce “pure” binary file, for this the ***struct*** module can be really useful
- See http://docs.python.org/lib/module-struct.html for more details.

- Alternatively you can use the function from VCDAT inside the “browser” module:

```
>>> browser.gui_read_Struct.read( file ,format="", \
    endian='@', datatype='f', ids=[], shape=[], \
    separator=""):
```

Reading Binary files

- Or you can use the contributed ‘binaryio’ package:

```
>>> from binaryio import *
>>> iunit = bincreate('filename')
>>> binwrite(iunit, some_array)
>>> # (the array can span 4 dimensions, or scalars)
>>> binclose(iunit)
>>> iunit = binopen('filename')
>>> y = binread(iunit, n, ...)   # (1-4 dimensions)
>>> binclose(iunit)
```

Self-Describing Binary Files (2)

- More recognised format are:
  - **GRIB** – is handled via the GrADS/GRIB interface, a slightly convoluted but effective way to get data into CDAT.
  - **PCMDI DRS** format – not covered here as relatively little UK usage.
  - **CDML** (Climate Data Markup Language) – the internal CDAT XML representation that points to multiple binary files.

Reading GRIB 1

To read GRIB (regular grids only), use the "**grib2ctl.pl**" perl script to generate the control file (".ctl").

```
dset ^test.grb
index ^test.grb.idx
undef 9.999E+20
title test.grb
*  produced by grib2ctl v0.9.12.5p32l
dtype grib 255
options yrev
ydef 181 linear -90.000000 1
xdef 360 linear 0.000000 1.000000
tdef 1 linear 18Z01jan1996 6hr
zdef 21 levels
21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
vars 1
O3hbl 60 203,109,0 ** Ozone mass mixing ratio kg kg**-1
```

Example Control (*.ctl) file

[produced by *grib2ctl.pl*]

***grib2ctl.pl* is available at:**

http://www.cpc.ncep.noaa.gov/products/wesley/grib2ctl.html

Reading GRIB 2

The ‘gribmap’ utility (part of GrADS) is used to create a small index file that points to the correct sections of the GRIB file to access the actual data.

Typical usage:

```
$ grib2ctl.pl afile.grb > afile.ctl
$ gribmap –e –i afile.ctl
# Open via the “afile.ctl” file.
```

***gribmap* is available as part of G^{r}ADS at:**

http://grads.iges.org/grads/

Other self-describing formats of interest in the UK

- You can also get support for:
  - **PP-format** – the BADC has developed code for reading the Met Office proprietary field data format. This should soon be included in the I/O layer beneath CDMS (known as cdunif – a C-layer that provides read access to multiple formats, and write access to NetCDF). Ask for details.
  - **NASA Ames** – a group of ASCII formats developed at NASA for field experiments and data exchange. Used extensively in UK atmospheric research. The BADC has developed a NASA Ames I/O Python package that links to cdms (see: http://home.badc.rl.ac.uk/astephens/software/nappy).

Writing data to files

- Writing ASCII files.
- Writing non-standard binary files.

Writing ASCII files

- Writing ASCII files is largely about your preferences as a file author:
  - Do you want any metadata retained?
  - Do you want to follow any standards or make up your own brand?
  - How do you want the data (and/or metadata) formatted?

Writing ASCII files

- The simple view is to open an ASCII file and write to it:

```
>>> outfile=open('my_output.txt', 'w') # opens file
>>> # Now write header
>>> outfile.write("Header: CHORDEX34 data\n")
>>> outfile.write("Time\tTemp (K)\tWspd (m/s)\n")
>>> # Get the time steps
>>> times=temp.getTime().asComponentTime()
>>> c=0
>>> while c<len(times): # Write data at each time
...   outstring="%s\t%s\t%s\n" % (times[c], \
                        temp[c], wspd[c])
...   outfile.write(outstring)
...   c=c+1
>>> outfile.close()
```

Writing ASCII files in NASA Ames format (1)

- The BADC has written a package to bridge the gap between the NASA Ames File format(s) developed in the 1990s for data exchange in scientific projects.

- **nappy – NASA Ames Processing in Python** – allows you to write CDMS variables directly to NASA Ames (some sub-formats will choke, but most will work!).

- Get nappy (beta-release) at:

  http://home.badc.rl.ac.uk/astephens/software/nappy

- Command-line usage:

```
$ cdms2na.py –i cdmsFile.nc –o naFile.na
```

Writing ASCII files in NASA Ames format (2)

Working with nappy and CDMS:

```
>>> import nappy,cdms # import modules
>>> # open file
>>> cdmsFile=cdms.open('mydatafile.nc')
>>> # Get variable
>>> cdmsVar=cdmsFile('n2o5')
>>> # Create the NASA Ames builder instance
>>> naBuilder=nappy.CdmsToNABuilder(cmdsVars)
>>> # Write output to NASA Ames file
>>> nappy.openNAFile("my_file.na", "w", \
            naBuilder.naDict)
```

Writing non-standard binary files

- Once again you can use the python **struct** module to write more complex binary output files. For simple binary arrays written to files you can use the built-in python I/O:

```
>>> outfile=open("binary_var.dat", "wb")
>>> x=N.array([2,4,6,8,9], 'f')
>>> outfile.write(x)
>>> outfile.close()
```

- If you have a multi-dimensional variable then you might need to write each row according to your own file format design.
- *But why not use NetCDF?*